

TECHNICAL FIELD

This invention generally relates to language modeling and, more particularly, to a method and apparatus for generating and managing a language model data structure.

BACKGROUND

Recent advances in computing power and related technology have fostered the development of a new generation of powerful software applications including web-browsers, word processing and speech recognition applications. The latest generation of web-browsers, for example, anticipate a uniform resource locator (URL) address entry after a few of the initial characters of the domain name have been entered. Word processors offer improved spelling and grammar checking capabilities, word prediction, and language conversion. Newer speech recognition applications similarly offer a wide variety of features with impressive recognition and prediction accuracy rates. In order to be useful to an end-user, these features must execute in substantially real-time. To provide this performance, many applications rely on a tree-like data structure to build a simple language model.

Simplistically, a language model measures the likelihood of any given sentence. That is, a language model can take any sequence of items (words, characters, letters, etc.) and estimate the probability of the sequence. A common approach to building a prior art language model is to utilize a prefix tree-like data structure to build an N-gram language model from a known training set of text.

The use of a prefix tree data structure (a.k.a. a suffix tree, or a PAT tree) enables a higher level application to quickly traverse the language model, providing the substantially real-time performance characteristics described above.

Simplistically, the N-gram language model counts the number of occurrences of a

1 particular item (word, character, etc.) in a string (of size N) throughout a text. The
2 counts are used to calculate the probability of the use of the item strings.
3 Traditionally, a tri-gram (N-gram where N=3) approach involves the following
4 steps:

- 5 (a) characters (C) are segmented into words (W) using a pre-defined
6 lexicon, wherein each W is mapped in the tree to one or more C's;
7 (b) predict the probability of a sequence of words ($W_1, W_2, \dots W_M$) from
8 the previous two words:

9
10
$$P(W_1, W_2, W_3, \dots W_M) \approx \prod P(W_i \mid W_{i-1}, W_{i-2}) \quad (1)$$

11

12 The N-gram language model is limited in a number of respects. First, the
13 counting process utilized in constructing the prefix tree is very time consuming.
14 Thus, only small N-gram models (typically bi-gram, or tri-gram) can practically be
15 achieved. Second, as the string size (N) of the N-gram language model increases,
16 the memory required to store the prefix tree increases by 2^N . Thus, the memory
17 required to store the N-gram language model, and the access time required to
18 utilize a large N-gram language model is prohibitively large for N-grams larger
19 than three (i.e., a tri-gram).

20 As a consequence of these computational and architectural limitations, prior
21 art implementations of N-gram language models tend to be very rigid. That is,
22 prior art N-gram language models tend to use a standard (small) lexicon, a
23 simplistic segmentation algorithm, and will typically only rely on the previous two
24 words to predict the current word (in a tri-gram model).

25 A small lexicon limits the ability of the model to identify words to those
contained in the lexicon. If a word is not in the lexicon, it does not exist as far as

1 the model is concerned. A simplistic segmentation algorithm typically errors in
2 favor of larger words over smaller words. Thus, the model is unable to accurately
3 predict smaller words contained within larger lexiconically acceptable strings.
4 Moreover, the lexicon and segmentation algorithm that converts the characters to
5 words may be error-prone (e.g., it is well accepted that all known segmentation
6 algorithms make errors), and that such errors are then propagated through the
7 model thereby limiting its accuracy and predictive attributes.

8 Finally, limiting the model to at most the previous two words for context (in
9 a tri-gram language model) is also limiting in that a greater context might be
10 required to accurately predict the likelihood of a word. The limitations on these
11 three aspects of the language model often result in poor predictive qualities of the
12 language model.

13 Thus, an improved method and apparatus for generating and managing a
14 language model data structure is required, unencumbered by the deficiencies and
15 limitations commonly associated with prior art language modeling techniques.
16 Just such a solution is provided below.

17 **SUMMARY**

18 This invention concerns a method and apparatus for generating and
19 managing a language model data structure. According to a first aspect of the
20 invention, a method is presented comprising assigning each of a plurality of
21 segments comprising a received corpus to a node in a data structure denoting
22 dependencies between nodes, and calculating a transitional probability between
23 each of the nodes in the data structure.
24

25 According to a second aspect of the invention, a data structure is presented
comprising a root node and a plurality of subordinate nodes, ultimately linked to

1 the root node, cumulatively comprising one or more sub-trees, wherein each
2 subordinate node of a sub-tree includes a measure of a Markov transition
3 probability between the node and another node. In addition to the Markov
4 transition probability, each node of a binary tree includes a compare bit, and an
5 indication of which item of the corpus the node is associated. It will be
6 appreciated by those skilled in the art that utilizing binary sub-trees greatly reduces
7 the overall complexity of the language model while utilize a dynamically allocated
8 context, thereby improving overall performance of the language model when
9 compared to the prior art.

10 According to one implementation, the data structure is generated and
11 maintained by one or more computer systems. Accordingly, a computer system is
12 presented to manage a language model data structure having a plurality of nodes,
13 the computer system comprising a controller and a memory subsystem, coupled
14 with and responsive to the controller. The memory subsystem is comprised of a
15 first level cache, a second level cache and an extended memory (e.g., one or more
16 disk files). The first level cache stores a first subset of the plurality of nodes,
17 which are often accessed. The second level cache is utilized as a node write buffer
18 to the memory subsystem. In this regard, the second level cache stores two or
19 more "write node" commands, and the controller combines the two or more node
20 write commands into a single, composite node write command. The extended
21 memory is used to store one or more files which contain language model data
22 structure nodes that are not often used and are not, therefore, retained in the first
23 level cache.

BRIEF DESCRIPTION OF THE DRAWINGS

The same reference numbers are used throughout the figures to reference like components and features.

Fig. 1 is a block diagram of a computer system incorporating the teachings of the present invention;

Fig. 2 is a block diagram of an example modeling agent which builds and manages an innovative language model data structure, according to one implementation of the present invention;

Fig. 3 is a graphical representation of an example dynamic order markov model (DOMM);

Fig. 4 is a graphical illustration of an example memory model for storing a DOMM tree;

Fig. 5 is a flow chart of an example method for building a DOMM tree;

Fig. 6 is a flow chart of an example method for writing a node to the DOMM tree;

Fig. 7 is a flow chart of an example method for reading a node from the DOMM tree;

Fig. 8 is a flow chart of an example method for evaluating the probability of a textual corpus using a DOMM tree; and

Fig. 9 is a storage medium with a plurality of executable instructions which, when executed, implement the innovative modeling agent of the present invention, according to an alternate embodiment of the present invention.

DETAILED DESCRIPTION

This invention concerns a method and apparatus for generating and managing a language model data structure. According to one aspect of the

invention, the language model data structure is developed without the limitations of a pre-defined lexicon or a segmentation algorithm. Rather, the invention described herein calculates the Markov transition probabilities between individual characters is developing a language model. Moreover, the present invention dynamically modifies the context utilized in calculating the probability of an item, and is not limited to the prior two words (like a tri-gram language model). Insofar as the context is dynamically generated, and the language model relies on the Markov transition probabilities between individual characters, the innovative language model described herein is referred to as a Dynamic Order Markov Model (DOMM).

In the discussion herein, the invention is described in the general context of computer-executable instructions, such as program modules, being executed by one or more conventional computers. Generally, program modules include routines, programs, objects, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, personal digital assistants, multiprocessor systems, microprocessor-based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. In a distributed computer environment, program modules may be located in both local and remote memory storage devices. It is noted, however, that modification to the implementations and the methods may be made without deviating from the spirit and scope of the claimed invention.

EXAMPLE COMPUTER SYSTEM

1 **Fig. 1** illustrates an example computer system 102 including an innovative
2 language modeling agent 104, which develops and maintains a language model
3 data structure using Markov transition probabilities between individual characters
4 of a corpus. It should be appreciated that although depicted as a separate, stand
5 alone application in Fig. 1, language modeling agent 104 may well be implemented
6 as a function of an application, e.g., word processor, web browser, speech
7 recognition system, etc. It will be evident, from the discussion to follow, that
8 computer 102 is intended to represent any of a class of general or special purpose
9 computing platforms which, when endowed with the innovative language
10 modeling agent (LMA) 104, implement the teachings of the present invention in
11 accordance with the first example implementation introduced above. It is to be
12 appreciated that although the language modeling agent is depicted herein as a
13 software application, computer system 102 may alternatively support a hardware
14 implementation of LMA 104 as well. In this regard, but for the description of
15 LMA 104, the following description of computer system 102 is intended to be
16 merely illustrative, as computer systems of greater or lesser capability may well be
17 substituted without deviating from the spirit and scope of the present invention.

18 As shown, computer 102 includes one or more processors or processing
19 units 132, a system memory 134, and a bus 136 that couples various system
20 components including the system memory 134 to processors 132.

21 The bus 136 represents one or more of any of several types of bus
22 structures, including a memory bus or memory controller, a peripheral bus, an
23 accelerated graphics port, and a processor or local bus using any of a variety of bus
24 architectures. The system memory includes read only memory (ROM) 138 and
25 random access memory (RAM) 140. A basic input/output system (BIOS) 142,
containing the basic routines that help to transfer information between elements

within computer 102, such as during start-up, is stored in ROM 138. Computer 102 further includes a hard disk drive 144 for reading from and writing to a hard disk, not shown, a magnetic disk drive 146 for reading from and writing to a removable magnetic disk 148, and an optical disk drive 150 for reading from or writing to a removable optical disk 152 such as a CD ROM, DVD ROM or other such optical media. The hard disk drive 144, magnetic disk drive 146, and optical disk drive 150 are connected to the bus 136 by a SCSI interface 154 or some other suitable bus interface. The drives and their associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules and other data for computer 102.

Although the exemplary environment described herein employs a hard disk 144, a removable magnetic disk 148 and a removable optical disk 152, it should be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, random access memories (RAMs) read only memories (ROM), and the like, may also be used in the exemplary operating environment.

A number of program modules may be stored on the hard disk 144, magnetic disk 148, optical disk 152, ROM 138, or RAM 140, including an operating system 158, one or more application programs 160 including, for example, the innovative LMA 104 incorporating the teachings of the present invention, other program modules 162, and program data 164 (e.g., resultant language model data structures, etc.). A user may enter commands and information into computer 102 through input devices such as keyboard 166 and pointing device 168. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input

devices are connected to the processing unit 132 through an interface 170 that is coupled to bus 136. A monitor 172 or other type of display device is also connected to the bus 136 via an interface, such as a video adapter 174. In addition to the monitor 172, personal computers often include other peripheral output devices (not shown) such as speakers and printers.

As shown, computer 102 operates in a networked environment using logical connections to one or more remote computers, such as a remote computer 176. The remote computer 176 may be another personal computer, a personal digital assistant, a server, a router or other network device, a network “thin-client” PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to computer 102, although only a memory storage device 178 has been illustrated in Fig. 1.

As shown, the logical connections depicted in Fig. 2 include a local area network (LAN) 180 and a wide area network (WAN) 182. Such networking environments are commonplace in offices, enterprise-wide computer networks, Intranets, and the Internet. In one embodiment, remote computer 176 executes an Internet Web browser program such as the “Internet Explorer” Web browser manufactured and distributed by Microsoft Corporation of Redmond, Washington to access and utilize online services.

When used in a LAN networking environment, computer 102 is connected to the local network 180 through a network interface or adapter 184. When used in a WAN networking environment, computer 102 typically includes a modem 186 or other means for establishing communications over the wide area network 182, such as the Internet. The modem 186, which may be internal or external, is connected to the bus 136 via a input/output (I/O) interface 156. In addition to network connectivity, I/O interface 156 also supports one or more printers 188. In

1 a networked environment, program modules depicted relative to the personal
2 computer 102, or portions thereof, may be stored in the remote memory storage
3 device. It will be appreciated that the network connections shown are exemplary
4 and other means of establishing a communications link between the computers
5 may be used.

6 Generally, the data processors of computer 102 are programmed by means
7 of instructions stored at different times in the various computer-readable storage
8 media of the computer. Programs and operating systems are typically distributed,
9 for example, on floppy disks or CD-ROMs. From there, they are installed or
10 loaded into the secondary memory of a computer. At execution, they are loaded at
11 least partially into the computer's primary electronic memory. The invention
12 described herein includes these and other various types of computer-readable
13 storage media when such media contain instructions or programs for implementing
14 the innovative steps described below in conjunction with a microprocessor or other
15 data processor. The invention also includes the computer itself when programmed
16 according to the methods and techniques described below. Furthermore, certain
17 sub-components of the computer may be programmed to perform the functions and
18 steps described below. The invention includes such sub-components when they
19 are programmed as described. In addition, the invention described herein includes
20 data structures, described below, as embodied on various types of memory media.

21 For purposes of illustration, programs and other executable program
22 components such as the operating system are illustrated herein as discrete blocks,
23 although it is recognized that such programs and components reside at various
24 times in different storage components of the computer, and are executed by the
25 data processor(s) of the computer.

Example Language Modeling Agent

Fig. 2 illustrates a block diagram of an example language modeling agent (LMA) 104, incorporating the teachings of the present invention. As shown, language modeling agent 104 is comprised of one or more controllers 202, innovative analysis engine 204, storage/memory device(s) 206 and, optionally, one or more additional applications (e.g., graphical user interface, prediction application, verification application, estimation application, etc.) 208, each communicatively coupled as shown. It will be appreciated that although depicted in Fig. 2 as a number of disparate blocks, one or more of the functional elements of the LMA 104 may well be combined. In this regard, modeling agents of greater or lesser complexity which nonetheless utilize Markov transition probabilities to statistically model a language may well be employed without deviating from the spirit and scope of the present invention.

As alluded to above, although depicted as a separate functional element, LMA 104 may well be implemented as a function of a higher level application, e.g., a word processor, web browser, speech recognition system, or a language conversion system. In this regard, controller(s) 202 of LMA 104 are responsive to one or more instructional commands from the dominant application to selectively invoke the features of LMA 104. Alternatively, LMA 104 may well be implemented as a stand-alone language modeling tool, providing a user with a user interface (208) to selectively implement the features to be discussed below.

In either case, controller(s) 202 of LMA 104 selectively invoke one or more of the functions of analysis engine 204 to generate and manage a language model data structure 224. Thus, except as configured to effect the teachings of the present invention, controller 202 is intended to represent any of a number of alternate control systems known in the art including, but not limited to, a

microprocessor, a programmable logic array (PLA), a micro-machine, an application specific integrated circuit (ASIC) and the like. In an alternate implementation, controller 202 is intended to represent a series of executable instructions to implement the logic described above.

As shown, the innovative analysis engine 204 is comprised a Markov probability calculator 212, a data structure generator 210 including a frequency calculation function 213, a lexicon generation function 214 and a dynamic segmentation function 216, and a data structure memory manager 218. Upon receiving an external indication, controller 202 selectively invokes an instance of the analysis engine 204 to develop, modify or utilize a language model. Unlike prior art techniques for generating a language model, analysis engine 204 develops a language model data structure fundamentally based on the Markov transition probabilities between individual items of a textual corpus (e.g., one or more sets of text). Moreover, as will be shown, analysis engine 204 utilizes as much data (referred to as “context” or “order” as is available to calculate the probability of an item string. In this regard, the language model of the present invention is aptly referred to as a Dynamic Order Markov Model (DOMM).

When invoked by controller 202 to establish a DOMM data structure, analysis engine 204 selectively invokes the data structure generator 210. In response, data structure generator 210 establishes a tree-like data structure to be populated with nodes denoting inter-node dependencies. As described above, the tree-like data structure is referred to herein as a DOMM data structure, or DOMM tree. Controller 202 receives the textual corpus and stores at least a subset of the textual corpus in memory 206 as a dynamic training set 222 from which the language model is to be developed. It will be appreciated that, in alternate embodiments, a predetermined training set may also be used.

Once the dynamic training set is received, at least a subset of the training set 222 is retrieved by frequency calculation function 213 for analysis. Frequency calculation function 213 identifies a frequency of occurrence for each item (character, letter, number, word, etc.) in the training set subset from which the Markov transition probability is calculated. Based on inter-node dependencies, data structure generator 210 assigns each item to an appropriate node of the DOMM tree, with an indication of the frequency value (C_i) and a compare bit (b_i).

The Markov probability calculator 212 calculates the probability of an item (character, letter, number, etc.) from a context (j) of associated items. More specifically, according to the teachings of the present invention, the Markov probability of a particular item (C_i) is dependent on as many previous characters as data “allows”, in other words:

$$P(C_1, C_2, C_3, \dots, C_N) \approx \prod P(C_i \mid C_{i-1}, C_{i-2}, C_{i-3}, \dots, C_j) \quad (2)$$

The number of characters employed as context (j) by Markov probability calculator 212 is a “dynamic” quantity that is different for each sequence of characters $C_i, C_{i-1}, C_{i-2}, C_{i-3}$, etc. According to one implementation, the number of characters relied upon for context (j) by Markov probability calculator 212 is dependent, at least in part, on a frequency value for each of the characters, i.e., the rate at which they appear throughout the corpus. More specifically, if in identifying the items of the corpus Markov probability calculator 212 does not identify at least a minimum occurrence frequency for a particular item, it may be “pruned” (i.e., removed) from the tree as being statistically irrelevant. According to one embodiment, the minimum frequency threshold is three (3).

As alluded to above, analysis engine 204 does not rely on a fixed lexicon *or* a simple segmentation algorithm (both of which tend to be error prone). Accordingly, analysis engine 204 selectively invokes a dynamic segmentation function 216 to segment items (characters or letters, for example) into strings (e.g., words). More precisely, segmentation function 216 segments the training set 222 into subsets (chunks) and calculates a cohesion score (i.e., a measure of the similarity between items within the subset). The segmentation and cohesion calculation is iteratively performed by segmentation function 216 until the cohesion score for each subset reaches a predetermined threshold.

The lexicon generation function 214 is invoked to dynamically generate and maintain a lexicon 220 in memory 206. According to one implementation, lexicon generation function 214 analyzes the segmentation results and generates a lexicon from item strings with a Markov transition probability that exceeds a threshold. In this regard, lexicon generation function 214 develops a dynamic lexicon 220 from item strings which exceed a pre-determined Markov transition probability taken from one or more language models developed by analysis engine 204. Accordingly, unlike prior art language models which rely on a known, fixed lexicon that is prone to error, analysis engine 204 dynamically generates a lexicon of statistically significant, statistically accurate item strings from one or more language models developed over a period of time. According to one embodiment, the lexicon 220 comprises a “virtual corpus” that Markov probability calculator 212 relies upon (in addition to the dynamic training set) in developing subsequent language models.

When invoked to modify or utilize the DOMM language model data structure, analysis engine 204 selectively invokes an instance of data structure memory manager 218. According to one aspect of the invention, data structure

memory manager 218 utilizes system memory as well as extended memory to maintain the DOMM data structure. More specifically, as will be described in greater detail below with reference to Figs. 6 and 7, data structure memory manager 218 employs a WriteNode function and a ReadNode function (not shown) to maintain a subset of the most recently used nodes of the DOMM data structure in a first level cache 224 of a system memory 206, while relegating least recently used nodes to extended memory (e.g., disk files in hard drive 144, or some remote drive), to provide for improved performance characteristics. In addition, a second level cache of system memory 206 is used to aggregate write commands until a predetermined threshold has been met, at which point data structure memory manager make one aggregate WriteNode command to an appropriate location in memory. Although depicted as a separate functional element, those skilled in the art will appreciate that data structure memory manager 218 may well be combined as a functional element of controller(s) 202 without deviating from the spirit and scope of the present invention.

Example Data Structure – Dynamic Order Markov Model (DOMM) Tree

Fig. 3 graphically represents a conceptual illustration of an example Dynamic Order Markov Model tree-like data structure 300, according to the teachings of the present invention. To conceptually illustrate how a DOMM tree data structure 300 is configured, Fig. 3 presents an example DOMM data structure 300 for a language model developed from the English alphabet, i.e., A, B, C, ...Z. As shown the DOMM tree 300 is comprised of one or more root nodes 302 and one or more subordinate nodes 304, each associated with an item (character, letter, number, word, etc.) of a textual corpus, logically coupled to denote dependencies between nodes. According to one implementation of the present invention, root

nodes 302 are comprised of an item and a frequency value (e.g., a count of how many times the item occurs in the corpus). At some level below the root node level 302, the subordinate nodes are arranged in binary sub-trees, wherein each node includes a compare bit (b_i), an item with which the node is associated (A, B, ...), and a frequency value (C_N) for the item.

Thus, beginning with the root node associated with the item B 306, a binary sub-tree is comprised of subordinate nodes 308-318 denoting the relationships between nodes and the frequency with which they occur. Given this conceptual example, it should be appreciated that starting at a root node, e.g., 306, the complexity of a search of the DOMM tree approximates $\log(N)$, where N is the total number of nodes to be searched.

As alluded to above, the size of the DOMM tree 300 may exceed the space available in the memory device 206 of LMA 104 and/or the main memory 140 of computer system 102. Accordingly, data structure memory manager 218 facilitates storage of a DOMM tree data structure 300 across main memory (e.g., 140 and/or 206) into an extended memory space, e.g., disk files on a mass storage device such as hard drive 144 of computer system 102. A logical and architectural model of the DOMM data structure 300 traversing such memory boundaries is provided with reference to Fig. 4.

Fig. 4 graphically represents a logical and architectural model of the operation of the data structure memory manager 218 according to one implementation of the present invention. As shown, the data structure memory manager 218 manages a virtual memory space comprised of main memory 406 (e.g., system memory 140 and/or 206) and extended memory 408 (e.g., disk files on hard drive 144) in order to store DOMM tree 300. According to one implementation, data structure memory manager 218 maintains a pointer (Index) to

memory locations where each node of DOMM tree 300 is maintained. Thus, whenever an access (read or write) to a node of DOMM is made (block 402), data structure memory manger 218 determines whether the node is stored in main memory 406 or extended memory 408 based, at least in part, on the Index value for the particular node (block 408).

As shown, data structure memory manager 218 maintains an array of N_{mem} nodes in main memory 406. Once the number of nodes exceeds the size of main memory 406 (N_{mem}), data structure memory manager 218 identifies the most active nodes and maintains such nodes in main memory 406, while the remaining (less active) nodes are maintained at extended memory 408. According to one implementation, data structure memory manager 218 maintains an “age” value for each DOMM node in memory which denotes a count of the number of accesses that the DOMM tree has been accessed without hitting the node. The N_{mem} nodes with the lowest age are maintained in main memory 406, while any remaining nodes are maintained in extended memory 408.

Operation and Implementation

Having introduced the functional and conceptual elements of the present invention with reference to Figs. 1-4, the operation of the innovative language modeling agent 104 will now be described with reference to Figs. 5-8.

Building DOMM Tree Data Structure

Fig. 5 is a flow chart of an example method for building a language model data structure, according to the teachings of the present invention. As alluded to above, language modeling agent 104 may be invoked directly by a user or a higher-level application. In response, controller 202 of LMA 104 selectively invokes an instance of analysis engine 204, and a textual corpus (e.g., one or more documents)

is loaded into memory 206 as a dynamic training set 222 and split into subsets (e.g., sentences, lines, etc.), block 502. In response, data structure generator 210 assigns each item of the subset to a node in data structure and calculates a frequency value for the item, block 504. According to one implementation, once data structure generator has loaded the subset into the data structure, frequency calculation function 213 is invoked to identify the occurrence frequency of each item within the training set subset.

In block 506, data structure generator determines whether additional subsets of the training set remain and, if so, the next subset is read in block 508 and the process continues with block 504. In alternate implementation, data structure generator 210 completely loads the data structure, a subset at a time. Once the data structure is completely loaded with the items of the training set, data structure generator 210 invokes frequency calculation function 213 to identify the frequency value of the items.

If, in block 506 data structure generator 210 has completely loaded the data structure 300 with items of the training set 222, data structure generator 210 may optionally prune the data structure, block 510. A number of mechanisms may be employed to prune the resultant data structure 300.

According to one implementation, illustrated in Fig. 5, data structure generator 210 evaluates each node of the resultant data structure, beginning with the most subordinate nodes, to determine if the item is statistically relevant to the language model, block 512. In one embodiment, the relevance threshold is set to three (3). Accordingly, in block 514, data structure generator 210 determines whether the frequency of the node (F_{node}) exceeds the threshold value. If not, the node is removed from the data structure, block 516. If the frequency does meet or exceed the threshold (514), or once a prior block has been removed (516), data

1 structure generator 210 determines whether additional nodes remain for pruning
2 analysis, block 518. If so, data structure generator 210 moves to the next node,
3 block 520, and the process continues with block 512. As shown, the iterative
4 pruning process 510 continues until all of the nodes have been analyzed.

6 **Example DOMM Data Structure Memory Management**

7 As alluded to above, with respect to Fig. 4, data structure memory manager
8 218 manages the DOMM data structure across main memory 406 and extended
9 memory 408. DOMM nodes are created or modified with a WriteNode command,
10 while DOMM nodes are read with a ReadNode command. In this regard, data
11 structure memory manager 218 manages the writing to and reading from main
12 memory 406 and extended memory 408 using the WriteNode and ReadNode
13 commands. Example implementations of writing to and reading from the DOMM
14 data structure will thus be presented with reference to Figs. 6 and 7.

15 **Fig. 6** is a flow chart of an example method for writing a node of the
16 language model data structure to memory. As shown, the method begins with
17 block 602, wherein data structure memory manager 218 receives a writenode
18 command. In block 604, data structure memory manager determines whether the
19 requested node is located in main memory 406 or extended memory 408. More
20 specifically, data structure memory manager 218 identifies the pointer (Index)
21 associated with the requested node, and determines whether it points to a memory
22 location outside of main memory 406. If the Index pointer is pointing to a location
23 within main memory 406 ($\text{Index} \leq N_{\text{mem}}$), data structure memory manager 218
24 executes the WriteNode command, writing to the identified node, block 606, and
25 the process ends.

If, in block 604, the Index pointer is pointing to extended memory, a further determination is made of whether the Index pointer points to a location within a level one (L1) cache (i.e., a L1 cache hit), block 608. If so, data structure memory manager writes the node to L1 cache, block 610. As alluded to above, data structure memory manager 218 maintains a record of activity for each of the nodes of the data structure. According to one implementation, an “age” is maintained denoting the number of cycles that the DOMM has been accessed without accessing a particular node. If a node is accessed, the age of the node is reset to zero (0). Thus, once the memory manager writes the node to L1 cache in block 610, data structure memory manager updates the age of the node to zero (0), and increments the age of the remaining nodes by 1, block 612.

If, in block 608, the Index pointer did not point to a node in L1 cache, a determination is made of whether L1 cache is full, block 614. If the L1 cache is not full, data structure memory manager 218 writes the node to L1 cache, block 616. Alternatively, if L1 cache is full (614), data structure memory manager 218 identifies the least recently used node (i.e., the node with the highest age value), block 618, and determines whether the LRU node (K) is “dirty”, block 620. A node in L1 cache is “dirty” if it has been modified by an application, but not yet written to the disk file or lower level cache where it resides. If the LRU node is not dirty (i.e., has not been updated and, therefore, a copy is saved in lower level cache or extended memory), data structure memory manager writes to L1 cache even if it means overwriting K, block 614. Alternatively, if node K is dirty, K is written to a second level (L2) cache in block 622, thereby making room for the “new” node in L1 cache and data structure memory manager writes the node to L1 cache, block 614.

location outside of main memory 406. If the Index pointer is pointing to a location within main memory 406 ($\text{Index} \leq N_{\text{mem}}$), data structure memory manager 218 executes the ReadNode command, reading the identified node from main memory 406, block 706, and the process ends.

If, in block 704, the Index pointer is pointing to extended memory, a further determination is made of whether the Index pointer points to a location within a level one (L1) cache (i.e., a L1 cache hit), block 708. If so, data structure memory manager reads the node from L1 cache, block 710. As alluded to above, data structure memory manager 218 maintains a record of activity for each of the nodes of the data structure. According to one implementation, an “age” is maintained denoting the number of cycles that the DOMM has been accessed without accessing a particular node. If a node is accessed, the age of the node is reset to zero (0). Thus, once the memory manager reads the node to L1 cache in block 710, data structure memory manager updates the age of the node to zero (0), and increments the age of the remaining nodes by 1, block 712.

If, in block 708, the Index pointer did not point to a node in L1 cache, a determination is made of whether L1 cache is full, block 714. If the L1 cache is not full, data structure memory manager 218 reads the node from L2 cache, leaving a copy in L1 cache, block 716. Alternatively, if L1 cache is full (714), data structure memory manager 218 identifies the least recently used node (i.e., the node with the highest age value), block 718, and determines whether the LRU node (K) is “dirty”, block 720. As described above, a node in L1 cache is “dirty” if it has been modified by an application, but not yet written to the disk file or lower level cache where it resides. If the LRU node is not dirty (i.e., has not been updated and, therefore, a copy is saved in lower level cache or extended memory), data structure memory manager reads the node from L2 cache leaving a copy in L1

cache even if it means overwriting K, block 714. Alternatively, if node K is dirty, K is written to a second level (L2) cache in block 722, thereby making room for the “new” node in L1 cache and data structure memory manager reads the node from L2 cache leaving a copy in L1 cache, block 714.

Once the ReadNode command has been executed in block 714, the age of the nodes are updated in block 712, as described above, and the process ends.

According to one embodiment, alluded to above, the L2 cache is utilized as a write buffer for nodes in extended memory. Thus, data structure memory manager 218 must be able to read a node from L2 cache or disk files of extended memory 408. An example method for reading from L2 cache 716 is provided in blocks 724-728.

As shown, the method begins with data structure memory manager determining whether the ReadNode from Index results in an L2 cache hit, block 724. If so, the node is read from L2 cache, block 726 and the process ends. Alternatively, if the ReadNode does not result in an L2 cache hit, the node is read from extended memory 408, as identified by Index, block 728.

Having provided an example method for building, writing to and reading from a DOMM data structure with reference to Figs. 5-7, and example method for utilizing the innovative DOMM data structure to predict the likelihood of an item string is presented with reference to Fig. 8.

Example Implementation Using a DOMM

Fig. 8 is a flow chart of an example method for predicting the probability of an item using the language model data structure of the present invention. That is, LMA 104 receives an external indication to predict the probability of a string, given an initial corpus subset of items (e.g., characters, letters, numbers, etc.). In response, controller 202 invokes an instance of Markov probability calculator,

1 which initializes a probability measure associated with the subset to one ($P_{\text{subset}}=1$),
2 block 802. In blocks 804 and 806, counters i and j are initiated to 1 and 0,
3 respectively. Recall, from the discussion above, that the “ j ” counter represents a
4 context, and that according to one aspect of the present invention, the statistically
5 prediction of a character, C_i , is premised on as many data points as data allows,
6 i.e., the “dynamic order” of the Dynamic Order Markov Model.

7 In block 808, controller 202 via data structure memory manager determines
8 whether C_{j+1}, \dots, C_i are located within the DOMM data structure 300. In this
9 regard, data structure memory manager attempts to locate nodes associated with
10 items C_{j+1}, \dots, C_i in main memory 406 and/or extended memory 408 of the stored
11 DOMM data structure. If the item string is located within the DOMM data
12 structure 300, then the context counter, j , is incremented by one in block 810 and
13 the process continues with block 808 wherein data structure memory manager
14 attempts to find this larger item string.

15 Once the data structure memory manager cannot locate the item string in the
16 DOMM data structure, block 808, Markov probability calculator 212 determines
17 that the longest prefix of i is C_j, \dots, C_i , and calculates the probability of C_i , given
18 the dynamically derived order (or context) of $C_{i-1}, C_{i-2}, \dots, C_j$, block 812. Once the
19 probability of the string is determined, block 812, the probability of the subset is
20 re-calculated, block 814. More specifically, the probability of the subset is
21 updated to reflect the new item string:

$$P_{\text{subset}} = P_{\text{subset}} \times P(C_i \mid C_{i-1}, C_{i-2}, \dots, C_j) \quad (3)$$

22
23
24
25 In block 816, a determination is made of whether all of the items of the
subset have been included in the probability calculation, i.e., whether $i \geq N$, where

1 N represents the number of items in the subset. If not, Markov probability
2 calculator 212 moves on to the next item by incrementing "i", block 818, and
3 resets the context count of j to 0, block 806, and the process continues with block
4 808. If, alternatively, all of the items (N) of the subset have been analyzed (816),
5 then Markov probability calculator 212 returns a probability measure of the subset,
6 block 820.

7 It should be appreciated, given the foregoing that the language modeling
8 agent 104 and the DOMM data structure overcome a number of the limitations and
9 deficiencies commonly associated with prior art language modeling techniques.
10 First, the DOMM is novel in that it does not rely on a known lexicon or rigid
11 segmentation algorithm. Accordingly, DOMM does not suffer from the
12 propagation errors commonly associated with errors in lexicon or segmentation.
13 Second, the DOMM data structure does not suffer from use of a fixed order
14 (context) that the N-gram language model employs, (e.g., tri-gram approach).
15 Rather, DOMM utilizes as much context as data allows, i.e., the dynamic order
16 quality of DOMM. Further, DOMM unifies the language model and the dynamic
17 lexicon by promoting item strings which meet a probability threshold to the
18 dynamic lexicon.

19 20 **Alternate Embodiments**

21 Fig. 9 is a block diagram of a storage medium having stored thereon a
22 plurality of instructions including instructions to implement the innovative
23 modeling agent of the present invention, according to yet another embodiment of
24 the present invention. In general, Fig. 9 illustrates a storage medium/device 900
25 having stored thereon a plurality of executable instructions 902 including at least a
subset of which that, when executed, implement the innovative modeling agent

1 116 of the present invention. When executed by a processor of a host system, the
2 executable instructions 902 implement the modeling agent to generate a statistical
3 language model representation of a textual corpus for use by any of a host of other
4 applications executing on or otherwise available to the host system.

5 As used herein, storage medium 900 is intended to represent any of a
6 number of storage devices and/or storage media known to those skilled in the art
7 such as, for example, volatile memory devices, non-volatile memory devices,
8 magnetic storage media, optical storage media, and the like. Similarly, the
9 executable instructions are intended to reflect any of a number of software
10 languages known in the art such as, for example, C++, Visual Basic, Hypertext
11 Markup Language (HTML), Java, eXtensible Markup Language (XML), and the
12 like. Moreover, it is to be appreciated that the storage medium/device 900 need
13 not be co-located with any host system. That is, storage medium/device 900 may
14 well reside within a remote server communicatively coupled to and accessible by
15 an executing system. Accordingly, the software implementation of Fig. 9 is to be
16 regarded as illustrative, as alternate storage media and software embodiments are
17 anticipated within the spirit and scope of the present invention.

18 Although the invention has been described in language specific to structural
19 features and/or methodological steps, it is to be understood that the invention
20 defined in the appended claims is not necessarily limited to the specific features or
21 steps described. Rather, the specific features and steps are disclosed as exemplary
22 forms of implementing the claimed invention.